TestSelector: Automatic Test Suite Selection for Student Projects

Filipe Marques[1,2](✉), António Morgado[1], José Fragoso Santos[1,2], and Mikoláš Janota[3]

[1] INESC-ID, Lisboa, Portugal
[2] Instituto Superior Técnico, University of Lisbon, Lisboa, Portugal
filipe.s.marque@tecnico.ulisboa
[3] Czech Technical University in Prague, Prague, Czechia

Abstract. Computer Science course instructors routinely have to create comprehensive test suites to assess programming assignments. The creation of such test suites is typically not trivial as it involves selecting a limited number of tests from a set of (semi-)randomly generated ones. Manual strategies for test selection do not scale when considering large testing inputs needed, for instance, for the assessment of algorithms exercises. To facilitate this process, we present TestSelector, a new framework for automatic selection of optimal test suites for student projects. The key advantage of TestSelector over existing approaches is that it is easily extensible with arbitrarily complex code coverage measures, not requiring these measures to be encoded into the logic of an exact constraint solver. We demonstrate the flexibility of TestSelector by extending it with support for a range of classical code coverage measures and using it to select test suites for a number of real-world algorithms projects, further showing that the selected test suites outperform randomly selected ones in finding bugs in students' code.

Keywords: Constraint-based test suite selection · Runtime monitoring · Code coverage measures

1 Introduction

Computer science course instructors routinely have to create comprehensive test suites to automatically assess programming assignments. It is not uncommon for these test suites to have to be created before students actually submit their solutions. This is, for instance, the case when students are allowed to submit their solutions multiple times with the selected tests being run each time and feedback given to the student. In typical algorithms courses, testing inputs must be large enough to ensure that the students' solutions have the required asymptotic complexity. In such scenarios, course instructors usually resort to semi-random test generation, selecting only a small number of the generated tests due to the limited computational resources of testing platforms. Hence, the included tests must be judiciously chosen. Manual strategies for test selection, however, do not scale for large testing inputs.

This paper presents TestSelector, a new framework for optimal test selection for student projects. With our framework, the instructor provides a canonical implementation of the project assignment, a set of generated tests T, and the number n of tests to be selected, and TestSelector determines a subset $T' \subseteq T$ of size n that maximises a given code coverage measure. By maximising coverage of the canonical solution, TestSelector provides relative assurances that most of the corner case behaviours of the expected solution are covered by the selected test suite. Naturally, the better the coverage measure, the better those assurances. Importantly, the best coverage measure is often project-specific, there being no silver bullet.

The main advantage of TestSelector over existing approaches [1,11,14,25] is that it is easily extensible with arbitrarily complex code coverage measures specifically designed for the project at hand. Unlike previous approaches, TestSelector does not require the targeted coverage measures to be encoded into the logic of an exact constraint solver. We achieve this by using as our optimisation algorithm, a specialised version of the recent Seesaw algorithm [12] for exploring the Pareto optimal frontier of a pair of functions. We demonstrate the flexibility of TestSelector by extending it with support for a range of classical code coverage measures and using it to select test suites for a number of real-world algorithms projects, further showing that the selected test suites outperform randomly selected ones in finding bugs in students' code.

The paper starts with Sect. 2 that overviews the TestSelector framework presenting its main modules and how they interact. Section 3 presents an experimental evaluation of the framework. Section 4 overviews related work and concludes the paper. An extended version of the paper can be found in [16].

2 TestSelector Overview

We give an overview of our approach for selecting optimal test suites for student projects. As illustrated in Fig. 1, the TestSelector framework receives three inputs: **(1)** the instructor's implementation for the project, which we refer to as the *canonical solution*; **(2)** a JSON configuration file with a description of the coverage measure to be used for test selection as well as the number of tests to be selected; and **(3)** an initial set of input tests, T. Given these inputs, TestSelector computes an optimal subset of tests, $T' \subseteq T$, that maximises the selected coverage measure for the chosen number of tests, n ($|T'| = n$). Due to the combinatorial nature of the problem and the sheer size of the search space, it is often the case that TestSelector is not able to find the optimal solution within the given time constraints. In such cases, it returns the best solution found so far. Our experimental evaluation indicates that this solution is typically not far from the optimal one.

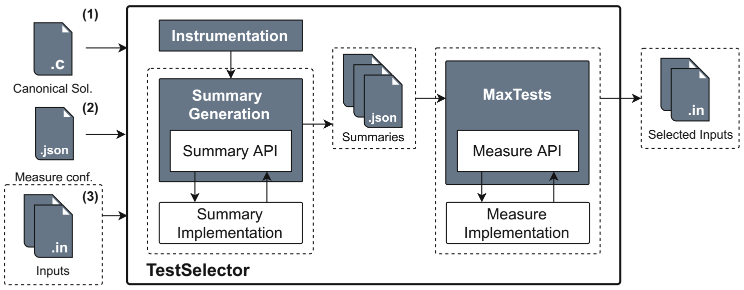

Fig. 1. TestSelector high-level architecture.

The TestSelector framework consists of two main building blocks:

- *Summary Generation Module:* The summary generation module automatically instruments the code of the canonical solution in order for its execution to additionally produce a *coverage summary* of each given input test. Different coverage measures require different summaries. For instance, a *block coverage summary* simply includes the identifiers of the code blocks that were executed during the running of the canonical solution.
- MaxTests *Module:* The MaxTests module receives as input the coverage measure to be used, the number n of tests to be selected, and a set of summaries, and selects the subset of size n of the given summaries that maximises the coverage measure. For instance, for the *block coverage measure*, MaxTests selects the summaries corresponding to the testing inputs that maximise the overall number of executed code blocks.
  At the core of MaxTests is an adapted implementation of the Seesaw algorithm [12], a novel algorithm for exploring the Pareto optimal frontier of two given functions using the well-known implicit hitting set paradigm [3, 4]. The key innovation of Seesaw is that it allows one to treat one of the two functions to optimise in a black-box manner. In our case, this black-box function corresponds to the targeted coverage function, meaning that we are able to select optimal test suites without encoding the targeted coverage functions into the logic of an exact constraint solver.

Supporting New Coverage Measures. The key advantage of TestSelector when compared to existing approaches for constraint-base test suite selection in the general setting [1,11,14,23,25] is that it is trivial to extend TestSelector with support for new, arbitrarily complex coverage measures. In contrast, existing approaches require users to encode the targeted coverage measures into the logic of an exact constraint solver, typically SMT [5] or Integer Linear Programming (ILP) solvers [10]. The manual construction of such encodings has two main inconveniences when compared to our approach. First, it requires expert knowledge of logic and inner workings of the targeted solver. Even simple encodings must be carefully engineered so that they can be efficiently solved. Second, there might be a mismatch between the expressivity of the existing solvers and the nature of the measure to be encoded. In contrast, with TestSelector, if one wants to add support for a new coverage measure, one simply has to:

1. Implement a *Coverage Summary API* that dynamically constructs a coverage summary during the execution of the canonical solution;
2. Implement a *Coverage Evaluation Function* that maps a given set of coverage summaries to a numeric coverage score. Importantly, in order for TestSelector to work properly, the coverage evaluation function must be *monotone*; meaning that for any two sets of summaries S_1 and S_2, it must hold that: $S_1 \subseteq S_2 \implies f(S_1) \leq f(S_2)$. Monotonicity is a natural requirement for coverage scoring functions.

Natively Supported Coverage Measures. Even though our main goal is to allow for users to easily implement their own coverage measures, TestSelector comes with built-in support for various standard code coverage measures. In particular, it implements: **(1)** *Block Coverage (BC)*—counts the number of executed code blocks; **(2)** *Array Coverage (AC)*—counts the number of programmatic interactions with distinct array indexes; **(3)** *Loop Coverage (LC)*—counts the number of loop executions with a distinct number of iterations; **(4)** *Decision Coverage (DC)*—counts the number of conditional guards that evaluate both to **true** and to **false**; **(5)** *Condition Coverage (CC)*—counts the number of conditional guards for which all subexpressions evaluate both to **true** and to **false**. We refer the reader to [22] for a detailed account of standard coverage measures in the software engineering literature.

Linear Combination of Coverage Measures. In addition to the coverage measures described above, TestSelector allows the user to specify a linear combination of coverage measures. Observe that, as the linear combination of two monotone functions is also monotone, the user is free to combine any monotone coverage measures without compromising the correct behaviour of MaxTests.

3 Evaluation

We evaluate TestSelector with respect to three research questions:

- **RQ1: How easy is it to extend TestSelector with new code coverage measures?** We show that the currently supported coverage measures are implemented with a small number of lines of code, demonstrating the practicality of our approach.
- **RQ2: Do classical code coverage measures improve test suite selection for bug finding in student projects?** We show that the test suites selected by TestSelector outperform randomly selected ones in finding bugs in students' code.
- **RQ3: Do linear combinations of code coverage measures further improve test suite selection for bug finding?** We show that by combining the best code coverage measures, we can find more bugs in students' code.

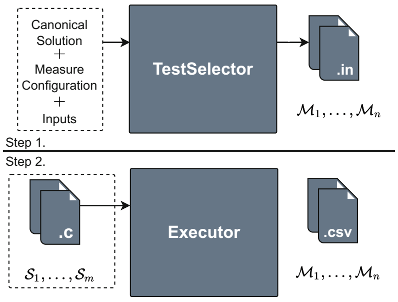
Canonical Solution + Measure Configuration + Inputs
TestSelector
.in
$\mathcal{M}_1, \dots, \mathcal{M}_n$
Step 1.
Step 2.
.c
$\mathcal{S}_1, \dots, \mathcal{S}_m$
Executor
.csv
$\mathcal{M}_1, \dots, \mathcal{M}_n$

Fig. 2. Evaluation diagram.

Table 1. Benchmark characterisation.

Project	$\mathcal{C}_{\text{LoC}}$	n_{proj}	T_{LoC}	Avg_{LoC}	n_{inpts}
P1	256	398	140,349	352.64	1,002
P2	529	349	176,547	505.86	600
P3	416	193	26,890	139.32	1,000
P4	208	166	34,512	207.90	1,000
P5	304	172	21,114	122.76	1,000
P6	204	185	24,091	130.22	800
P7	108	174	24,035	138.13	1,000
Total	2,125	1,637	447,538	273.39	6,402

Experimental Procedure. The experimental procedure is a two-step process, as illustrated in Fig. 2. In the first step, TESTSELECTOR selects the test suites for a given canonical solution, set of inputs, and configuration file specifying the coverage measures and the size of the computed test suites. This step generates a set of test suites, each corresponding to one of the specified measures. In the second step, an executor will run every student's project against the selected test suites. In the end, the executor creates a report detailing the passing/failing rate for every student's project on each selected test suite.

All the experiments were performed on a server with a 12-core Intel Xeon E5–2620 CPU and 32GB of RAM running Ubuntu 20.04.2 LTS. For the ILP solver we used the Gurobi Optimizer v9.1.2. For each execution of MAXTESTS we set a time limit of 30 min.

Benchmarks. We curated a benchmark suite comprising students' projects from seven editions of two programming courses organised by the authors. Table 1 presents the benchmark suite characterisation. For each project, we show the number of lines of code of the canonical solution ($\mathcal{C}_{\text{LoC}}$), the number of student projects (n_{proj}), the total number of lines of code of the student projects (T_{LoC}), the average number of lines of code per student project (Avg_{LoC}), and the number of available input tests (n_{inpts}). In summary, we tested 1,637 projects, which totalled 447K lines of code ($\approx$ 273 LoC/project).

3.1 RQ1: TESTSELECTOR Extensibility

The table below presents the number of lines of code of the implementation of each coverage measure: *Loop Coverage* (LC), *Array Coverage* (AC), *Block Coverage* (BC), *Condition Coverage* (CC), and *Decision Coverage* (DC). For each measure, we give the number of lines of code of both its implementation of the coverage summary API and evaluation function.

Module	LC	AC	BC	CC	DC
Coverage Summary API	90	60	42	120	120
Measure Evaluation Function	54	58	48	74	64

Table 2. Results for each measure with linear search (LS) and progression search (PS).

Project	Search	LC	AC	BC	Size	CC	DC	Rnd
P1	LS	**14.67**	14.48	13.69	0.20	13.95	14.41	4.81
	PS	**14.53**	14.34	13.56	0.19	13.82	14.27	
P2	LS	18.07	17.15	**19.47**	6.14	15.60	14.35	5.60
	PS	18.07	17.20	**19.47**	6.14	15.60	14.35	
P3	LS	16.39	20.38	7.49	**28.07**	7.49	7.49	7.56
	PS	16.77	20.70	7.95	**28.31**	7.95	7.95	
P4	LS	**23.68**	22.78	11.99	23.59	17.95	17.93	13.52
	PS	**23.68**	22.82	11.99	23.59	17.93	17.93	
P5	LS	**3.76**	3.23	3.56	3.74	3.56	3.56	3.09
	PS	**3.76**	3.25	3.56	3.74	3.56	3.56	
P6	LS	6.91	8.22	8.01	**8.39**	4.72	4.68	6.61
	PS	6.91	8.22	8.01	**8.39**	4.72	4.66	
P7	LS	**10.46**	6.08	6.71	6.39	7.17	7.17	6.28
	PS	**10.46**	6.08	6.71	6.39	7.17	7.17	
Average	LS	**13.42**	13.19	10.13	10.93	10.06	9.94	6.78
	PS	**13.45**	13.23	10.18	10.96	10.11	9.98	

When it comes to the implementation of the coverage summary API, we observe that the simpler coverage measures, such as LC, AC, and BC require fewer than 100 lines of code to implement and the more complex coverage measures, such as CC and DC, require 120 lines of code. As expected, the measure evaluation function is simpler to implement than the coverage summary API, requiring even fewer lines of code (between 48–74 LoC).

3.2 RQ2: Classical Code Coverage Selection

We investigate the effectiveness of TestSelector when used to select test suites for finding bugs in students' code. In particular, we compare the number of bugs found by the test suites selected by TestSelector against those found by test suites obtained through random selection. In all experiments, we ask for test suites of size 30 out of 900 available randomly generated tests (the number of tests used to assess the students in the corresponding courses was 30). We consider the five coverage measures described in Sect. 2 and an additional measure corresponding to the size of the testing input. Furthermore, we run the Seesaw algorithm with two complementary search strategies: linear search (LS) and progression search (PS). Details can be found in [12,16].

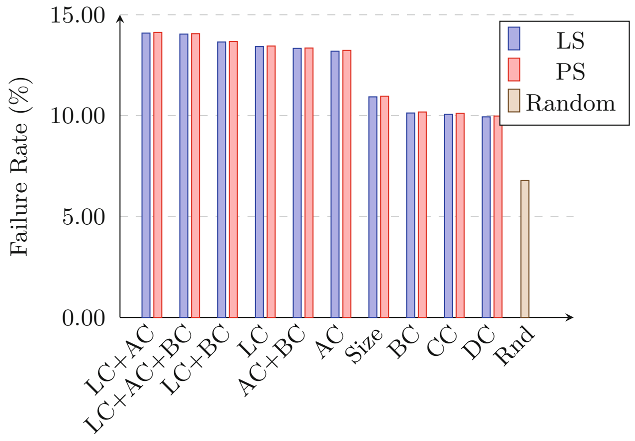

Fig. 3. Failure rate (%) for each measure, comparing linear search (LS) with progression search (PS).

Results. Table 2 presents the results of the experiment. For each project, the table shows the resulting failure rates for the measures *Loop Coverage* (LC), *Array Coverage* (AC), *Block Coverage* (BC), *Size*, *Condition Coverage* (CC), and *Decision Coverage* (DC). We observe that the best measure is project-dependent, with LC being the best measure in four projects, BC in one, and Size in two. Importantly, we also observe that the more sophisticated measures, such as CC and DC, have lower failure rates than simpler measures, such as LC and AC. This may be explained by the fact that the students' most common programming errors are often encoded in loops and array accesses. All coverage measures consistently perform better than the random test suite selection.

3.3 RQ3: Linear Combinations of Coverage Measures

To investigate whether using linear combinations of code coverage measures can further improve the bug finding results, we replay the experiment described in Sect. 3.2 with the following combinations of coverage measures: **(1)** AC+LC; **(2)** BC+LC; **(3)** AC+BC; and **(4)** AC+BC+LC.

Results. Figure 3 presents the obtained results for the four linear combinations[1] and the five individual code coverage measures presented in Table 2. For each measure, we give a blue and a red bar, each corresponding to one of the search strategies supported by the Seesaw algorithm. It is easy to observe that the majority of the combinations, i.e., LC+AC, LC+AC+BC, and LC+BC, are able to find more bugs in the students' code than the overall best-performing single measure (LC), with only AC+BC obtaining worse results.

[1] LC+AC, LC+BC, AC+BC, and LC+AC+BC.

4 Related Work and Conclusions

Test Suit Construction. The software engineering community has dedicated a considerable effort to the problem of generating effective test suites for complex software systems, exploring topics such as: test suite reduction and test case selection [1,2,13,14,18,26], combinatorial testing [23–25], and a variety of fuzzying strategies [6–9,19]. In the following, we focus on the test suite reduction and test case selection problems, which are immediately close to our own goal, highlighting constraint-based approaches. Importantly, we are not aware on any works in this field specifically targeted at student projects. The testing of such projects has, however, its own specificities when compared to the testing of large-scale industrial software systems. In particular, the time constraints on the test generation process are less severe and the code being tested less complex.

The *test suite reduction problem* [1,2,17,21,26] aims at reducing the size of a given test suite while satisfying a given test criterion. Typical criteria are the so-called coverage-based criteria, which ensure that the coverage of the reduced test suite is above a certain minimal threshold. The *test case selection problem* [1,2,17,21,26] is the dual problem, in that it tries to determine the minimal number of tests to be added to a given test suite so that a given test criterion is attained. As most of these algorithms target industrial settings, they assume severe time constraints on the test selection process. Hence, the vast majority of the proposed approaches for test suite reduction and selection are approximate, such as similarity-based algorithms [2,17], which are not guaranteed to find the optimal test suite even when given enough resources. In order to achieve a compromise between precision and scalability, the authors of [1] proposed a combination of standard ILP encodings and heuristic approaches. Finally, the authors of [14] proposed a SAT-based encoding for selecting optimal test suites according to the modified condition decision coverage criterion [13,22]. They argue that, as this criterion is enforced by safety standards in both the automative and the avionics industries, one is obliged to resort to exact approaches.

Conclusions and Future Work. We have presented TestSelector, a new framework for the automatic selection of optimal test suites for student projects. The key innovation of TestSelector is its extensibility to support new code coverage measures without these measures being encoded into the logic of an exact constraint solver. We evaluate TestSelector against a benchmark comprised of 1,637 real-world student projects, demonstrating that: **(1)** it is trivial to extend TestSelector with support for new coverage measures and **(2)** the selected test suites outperform randomly selected ones in finding bugs in students' code.

In the future, we plan to conduct a more thorough investigation on the relation between the characteristics of a project and the code coverage measures that are appropriate for it. We also plan to integrate TestSelector with an existing testing platform for student projects, such as Mooshak [15] or Pandora [20].

Acknowledgements. The authors were supported by Portuguese national funds through Fundação para a Ciência e a Tecnologia (UIDB/50021/2020, INESC-ID multiannual funding program) and projects INFOCOS (PTDC/CCI-COM/32378/2017) and DIVINA (CMU/TIC/0053/2021). The results were also supported by the MEYS within the dedicated program ERC CZ under the project *POSTMAN* no. LL1902, and it is part of the *RICAIP* project that has received funding from the European Union's Horizon 2020 under grant agreement No 857306.

References

1. Chen, Z., Zhang, X., Xu, B.: A degraded ILP approach for test suite reduction. In: Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE), pp. 494–499. Knowledge Systems Institute Graduate School (2008)
2. Cruciani, E., Miranda, B., Verdecchia, R., Bertolino, A.: Scalable approaches for test suite reduction. In: Proceedings of the 41st International Conference on Software Engineering, ICSE, pp. 419–429. IEEE / ACM (2019)
3. Davies, J., Bacchus, F.: Solving MaxSAT by solving a sequence of simpler SAT instances. In: Principles and Practice of Constraint Programming (2011)
4. Davies, J., Bacchus, F.: Exploiting the power of MIP solvers in MaxSAT. In: Theory and Applications of Satisfiability Testing (2013)
5. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems (2008)
6. Godefroid, P.: Compositional dynamic test generation. In: POPL, vol. 42, pp. 47–54 (2007)
7. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: ACM Sigplan Notices (2005)
8. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS (2008)
9. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: Unleashing the power of alternation. In: POPL (2010)
10. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2022). https://www.gurobi.com
11. Hnich, B., Prestwich, S.D., Selensky, E., Smith, B.M.: Constraint models for the covering test problem. Constraints An Int. J. **11**(2–3), 199–219 (2006)
12. Janota, M., Morgado, A., Fragoso Santos, J., Manquinho, V.: The Seesaw Algorithm: Function Optimization Using Implicit Hitting Sets. In: Principles and Practice of Constraint Programming (2021)
13. Jones, J.A., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Software Eng. **29**(3), 195–209 (2003)
14. Kitamura, T., Maissonneuve, Q., Choi, E.-H., Artho, C., Gargantini, A.: Optimal test suite generation for modified condition decision coverage using SAT solving. In: Gallina, B., Skavhaug, A., Bitsch, F. (eds.) SAFECOMP 2018. LNCS, vol. 11093, pp. 123–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99130-6_9
15. Leal, J.P., Paiva, J.C., Correia, H.: Mooshak (2022). https://mooshak2.dcc.fc.up.pt
16. Marques, F., Morgado, A., Santos, J.F., Janota, M.: TestSelector: automatic test suite selection for student projects - extended version (2022). https://doi.org/10.48550/ARXIV.2207.09509. https://arxiv.org/abs/2207.09509

17. Miranda, B., Cruciani, E., Verdecchia, R., Bertolino, A.: FAST approaches to scalable similarity-based test case prioritization. In: Proceedings of the 40th International Conference on Software Engineering, ICSE, pp. 222–232. ACM (2018)
18. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (eds.) SSBSE 2015. LNCS, vol. 9275, pp. 93–108. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22183-0_7
19. Sen, K., Agha, G.: Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: CAV, pp. 419–423 (2006)
20. Serra, P.: Pandora: Automatic Assessment Tool (AAT) (2022). https://saturn.ulusofona.pt
21. Shi, A., Yung, T., Gyori, A., Marinov, D.: Comparing and combining test-suite reduction and regression test selection. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, pp. 237–247. ACM (2015)
22. Szũgyi, Z., Porkoláb, Z.: Comparison of DC and MC/DC code coverages. Research report, Acta Electrotechnica et Informatica (2013)
23. Wu, H., Nie, C., Petke, J., Jia, Y., Harman, M.: A survey of constrained combinatorial testing. CoRR abs/1908.02480 (2019)
24. Yamada, A., Biere, A., Artho, C., Kitamura, T., Choi, E.: Greedy combinatorial test case generation using unsatisfiable cores. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE, pp. 614–624. ACM (2016)
25. Yamada, A., Kitamura, T., Artho, C., Choi, E., Oiwa, Y., Biere, A.: Optimization of combinatorial testing by incremental SAT solving. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST, pp. 1–10. IEEE Computer Society (2015)
26. Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, pp. 140–150. ACM (2007)